Hardware Support for Extracting ILP

Prof. Naga Kandasamy
ECE Department
Drexel University

May 8, 2016

These notes are derived from:

- D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 2th Edition, Morgan Kaufmann, 1996.

We discuss some hardware-based approaches to extract ILP.

- Extension of the ISA to include conditional or predicated instructions.
- Avoiding control dependence stalls by allowing execution of an instruction before the processor knows that the instruction should execute. This idea is called speculation.
  - Static speculation performed by the compiler with hardware support.
  - Dynamic speculation in hardware.

1 ISA Support: Conditional or Predicated Instructions

Consider the following code:

```
if (A == 0)
    X = Y;
```

Assuming registers `R1`, `R2`, and `R3` hold the values for `A`, `X`, and `Y`, respectively, a straightforward code using a branch statement is as follows.

```
   BNEZ R1, L ; Check if A = 0
   MOV R2, R3 ; X = Y
L:
```

We can implement the above code snippet in one *conditional instruction*, in this case, a conditional move that performs the move only if the third operand is equal to zero as follows:

```
CMOVZ   R2, R3, R1
```

For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline where the register write occurs.

As another example, consider the absolute value `A = abs(B)`. This is implemented as:

```
if (B < 0)
   A = -B;
else
   A = B;
```

Assuming that registers `R1` and `R2` contain the variables `A` and `B`, respectively, the MIPS code for the above snippet is as follows:

```
    SLT  R2, $zero, R3   ; R3 is 1 if R2 < 0, otherwise R3 is 0
    BNEZ R3, L1
    ADD  R1, R2, $zero   ; A = B
    J    L2
L1: SUB  R1, $zero, R2   ; A = -B
L2:
```

The above code can be implemented using a conditional move.

```
SUB   R1, $zero, R2   ; A = -B (the unconditional move)
SLT   R2, $zero, R3   ; Check if B < 0
CMOVZ R1, R2, R3      ; A = B only if B >= 0
```

A conditional instruction can also be used to speculatively move an instruction across a branch statement to improve instruction scheduling. For example, consider the following code sequence for a two-issue superscalar MIPS processor that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

Slot 1	**Slot 2**
`LW R1, 40(R2)`	`ADD R3, R4, R5`
	`ADD R6, R3, R7`
`BEQZ R10, L`	
`LW R8, 20(R10)`	
`LW R9, 0(R8)`	

Assume that we have a conditional version of load word called `LWC` and assume that the load occurs unless the third operand is 0. Now, we can speculatively issue the load immediately following the `BEQZ` as follows:

Slot 1	**Slot 2**
`LW R1, 40(R2)`	`ADD R3, R4, R5`
`LWC R8, 20(R10), R10`	`ADD R6, R3, R7`
`BEQZ R10, L`	
`LW R9, 0(R8)`	

This improves the execution time since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. If the compiler mispredicts the branch, the conditional instruction will have no effect and will not improve the running time. This is why the code transformation is speculative.

To use a conditional instruction successfully in examples like this, we must ensure that the speculated instruction does not introduce an exception. A conditional instruction should have not effect on the pipeline state if the condition is not satisfied, meaning that the instruction cannot write the result destination nor cause any exception if the condition is not satisfied. For example, if `R10` contains zero, the instruction `LW R8, 20(R10)` is likely to cause a segmentation fault and this exception should not occur. In fact, this is the property that prevents the compiler from moving the load of `R8` across the branch. If `R10` does not contain zero, the `LW` may still cause a legal and resumable exception—for example, a page fault. So, the hardware must take the exception when it knows that the controlling condition is true.

Conditional instructions are helpful for implementing short alternative control flows such as in an if-then with a small then body.

2 Static Compiler Speculation with Hardware Support

When moving instructions across a branch, the compiler must ensure that exception behavior is not changed and that the dynamic data dependencies remain the same. We will discuss three methods for supporting speculative execution without introducing erroneous exception behavior:

1. The hardware and the operating system cooperatively ignore exceptions for speculative instructions.
2. A set of status bits, called *poison* bits, are attached to the result registers written by speculated instructions when these instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.
3. A mechanism is provided to indicate that an instruction is speculative and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

We need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed and can be accepted and processed for speculative instructions just as if they were normal instructions. If exceptions indicating a program error arise in speculative instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest case, the hardware and the Operating System simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause an exception—the so called silent exception. Consider the following code fragment:

```
if (A == 0)
    A = B;
else
    A = A + 4;
```

where A is at 0(R3) and B is at 0(R2). The corresponding assembly code is

```
        LW    R1, 0(R3)    ; Load A
        BNEZ  R1, L1       ; Test A
        LW    R1, 0(R2)    ; if clause
        J     L2
L1:     ADD   R1, R1, 4    ; else clause
L2:     SW    0(R3), R1    ; Store A
```

Assume the then clause is *almost always* executed.[1] The new code in which register B is speculatively loaded is as follows:

```
      LW    R1, 0(R3)    ; Load A
      LW    R14, 0(R2)   ; Speculative load B
      BEZ   R1, L3       ; Test A
      ADD   R14, R1, 4   ; else clause
L3:   SW    0(R3), R14   ; Nonspeculative store A
```

Notice that the then clause is completely speculated. We introduce a temporary register to avoid clobbering register R1 when B is loaded. Also note that the else clause could have been compiled speculatively with a conditional move.

Register renaming is often needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed or clobbered, and stores cannot be speculative.

In terms of handling exceptions, under the ”silent” exception scheme, it is not necessary to know that an instruction is speculative. If the instruction generating the exception is speculative, in this case, the load into register R14, returning an undefined value for the instruction cannot be harmful. If on the other hand, this was a useful instruction to execute, the incorrect program which formerly might have received a terminating exception will get an incorrect result. So, this scheme can never cause a correct program to fail.

The use of *poison* bits allows compiler speculation to be performed with less change to exception behavior. In particular, incorrect programs that caused termination without speculation will still cause exceptions when instructions are speculated. A poison bit is added to every register and

[1]GCC allows programmers to annotate the expected value of an expression—for example, to tell the compiler whether a conditional statement is likely to be true or false.

```
#define likely(x)    __builtin_expect (!!(x), 1)
#define unlikely(x)  __builtin_expect (!!(x), 0)
```

Programmers can mark an expression as likely or unlikely true by wrapping it in likely() or unlikely(), respectively. The following example marks a branch as unlikely true (likely to be false):

```
      int ret;
      ...
      ret = close (fd);    // fd is a file descriptor
      if (unlikely (ret))
          perror ("close");
```

Note that close() returns zero on a successful completion, no zero otherwise.

another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with the poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a source register with the poison bit turned on, the instruction causes a fault.

Consider the following code from earlier.

```
      LW    R1, 0(R3)    ; Load A
      BNEZ  R1, L1       ; Test A
      LW    R1, 0(R2)    ; if clause
      J     L2
L1:   ADD   R1, R1, 4    ; else clause
L2:   SW    0(R3), R1    ; Store A
```

The above code can be compiled as follows using speculation and poison bits.

```
       LW    R1, 0(R3)    ; Load A
       LW*   R14, 0(R2)   ; Speculative load B
       BEZ   R1, L3       ; Test A
       ADD   R14, R1, 4   ; else clause
L3:    SW    0(R3), R14   ; exception-speculative store
```

If the speculative `LW*` causes an exception, the poison bit of `R14` will be turned on. When the nonspeculative `SW` instruction occurs, it will raise an exception if the poison bit for `R14` is on.

The main disadvantage of the above-discussed schemes is the need to introduce copies to deal with register renaming, leading to the possibility of exhausting the register file. An alternative is to move instructions across branches, flagging them as speculative, and providing renaming and buffering in the hardware, much as Tomasulo's algorithm does. This concept is called *boosting*.

Returning to our code example, we use ”+” after the opcode to indicate that the instruction has been boosted across the branch.

```
      LW    R1, 0(R3)    ; Load A
      LW+   R1, 0(R2)    ; Boosted load B
      BEZ   R1, L3       ; Test A
      ADD   R1, R1, 4    ; else clause
L3:   SW    0(R3), R1    ; nonspeculative store
```

The extra register (`R14` in the previous examples) is no longer necessary, since if the branch is not taken the result of the speculative load is never written to `R1`; so `R1` can be used in the code sequence. Also, the result of the boosted load is not written to `R1` until after the branch outcome is known. Hence the branch uses the value of `R1` produced by the first nonboosted load. Other boosted instructions could use the results of the boosted load.

Again, the important point is that when the branch following the boosted instruction is reached, if the boosted instruction contained the correct prediction of the branch, the results are committed to the registers, else the results are discarded. So, boosted instructions allow the execution of an instruction that is dependent on a branch before the branch is resolved, but the final action to commit the instruction is taken only after the branch is resolved.

Hardware-based Speculation

Hardware-based speculation combines three major ideas: dynamic branch prediction to choose which instructions to execute; speculation to allow the execution of instructions before the control dependencies are resolved; and dynamic scheduling of instructions. This method of execution is called *data flow* execution: operations execute as soon as their operands are available.

Hardware-based speculation has the following advantages over compiler-based methods:

- Dynamic run-time disambiguation of memory addresses can be done using methods used for Tomasulo's algorithm. This allows us to move loads past stores at run time (if the load and store operations do not access the same memory location). This is difficult to do at compile time for programs that contain pointers.
- Hardware-based speculation is better in cases in which hardware-based branch prediction is better than prediction done at compile time.
- One can maintain a completely precise exception model even for speculated instructions.

The major disadvantage of hardware-based speculation is the substantial hardware resources needed to support it.

The hardware that implements Tomasulo's algorithm can be extended to support speculation. Here we separate the bypassing/forwarding of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to forward its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative. Using the bypass/forwarding paths then is like a speculative register read since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory. This is called as *committing* the instruction.

The key ideas behind speculation are the following:

- Allow instructions to execute out of order but force them to commit *in order* to prevent any irrevocable action such as updating the pipeline state or taking an exception.
- Separate the process of completing execution from instruction commit.

To support speculative execution, we need an additional set of hardware buffers, called a reorder buffer, to hold the results of instructions that have finished execution but have not committed. The reorder buffer is also used to pass results among instructions that may be speculated.

The *reorder buffer* provides additional virtual registers in the same way the reservation stations in Tomasulo's algorithm extend the register set. The buffer holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction

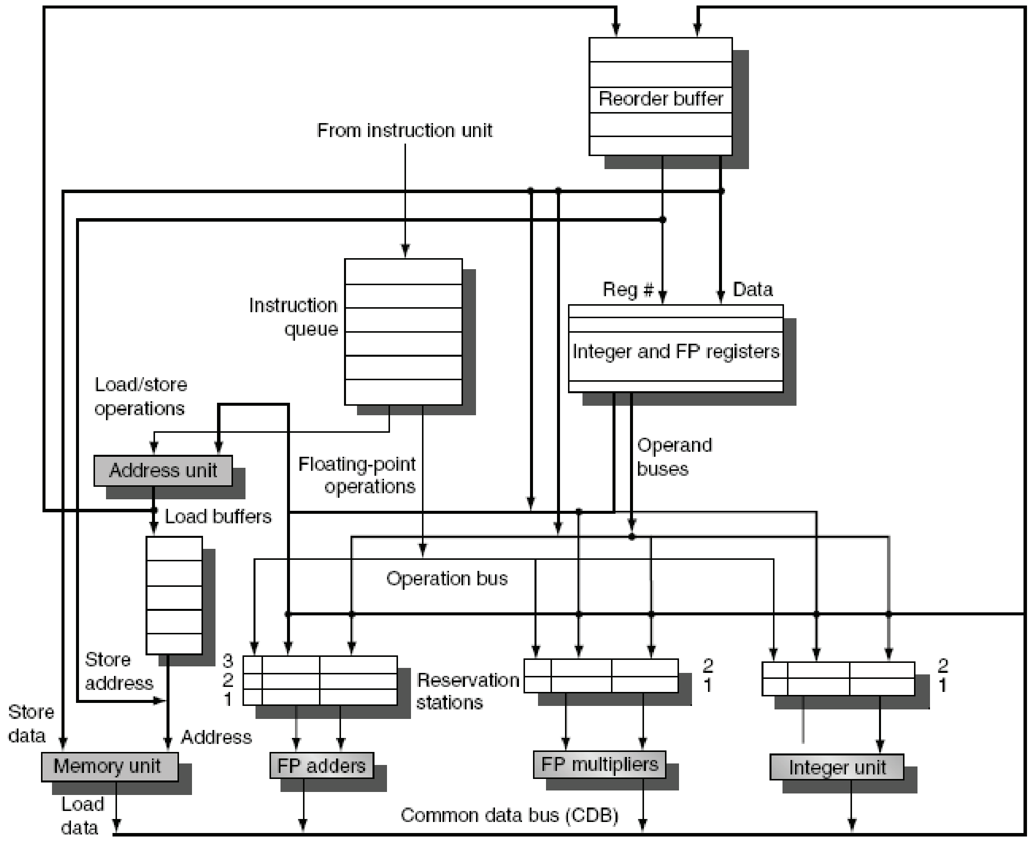

Figure 1: Basic structure of the pipeline that performs hardware speculation by extending the Tomsasulo algorithm. The major changes are the addition of the reorder buffer and the elimination of the load and store buffers since their functions are subsumed by the reorder buffer.

commits. Hence, the reorder buffer is a source of operands for instructions just as the reservation stations provide operands in Tomasulo's algorithm—the key difference being, in Tomasulo's algorithm once an instruction writes its result, subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits (we know definitely that the instruction should execute). Since the reorder buffer is responsible for holding results until they are stored into the registers, it replaces the functions of the load and store buffers in Tomasulo.

Each entry in the reorder buffer contains three fields: the instruction type, the destination field, and the value field. The instruction type field indicates whether it is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which have register destinations). The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores), where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits.

Figure 1 shows the basic structure of a processor supporting hardware speculation. The reorder buffer completely replaces the load and store buffers. Although the renaming function of the reservation stations is replaced by the reorder buffer, we still need a place to buffer operations (and operands) between the time they issue and the time they begin execution. This function is still provided by the reservation stations. Since every instruction has a position in the reorder buffer until it commits (and the results are posted to the register file), we tag a result using the reorder buffer entry number rather than using the reservation station number. This requires that the reorder buffer assigned for an instruction must be tracked in the reservation stations.

We now outline the basic steps involved in executing an instruction:

Issue: Get an instruction from the floating-point operation queue. Issue the instruction if there is an empty reservation station and an empty slot in the reorder buffer, send the operands to the reservation station if they are in the registers or the reorder buffer, and update the control entries to indicate the buffers are in use. The number of the reorder buffer allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the common data bus (CDB). If either all reservations are full or the reorder buffer is full, then instruction issue is stalled until both have available entries. This stage is sometimes called *dispatch* in a dynamically scheduled machine.

Execute: If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation.

Write result: When the result is available, write it on the CDB (with the reorder buffer tag sent when the instruction issued) and from the CDB into the reorder buffer, as well as to any reservation stations waiting for this result. Mark the reservation station as available.

Commit: When an instruction, other than a branch with incorrect prediction, reaches the head of the reorder buffer and its result is present in the buffer, update the register with the result (or perform a memory write if the operation is a store and remove the instruction from the reorder buffer. When a branch with an incorrect prediction reaches the head of the reorder buffer, it indicates that the speculation was wrong. The reorder buffer is flushed and execution is restarted at the correct successor of the branch.

Once an instruction commits, its entry in the reorder buffer is reclaimed and the register or memory destination is updated, eliminating the need for the reorder buffer entry. To avoid changing the reorder buffer numbers as instructions commit, we implement the reorder buffer as a circular queue, so that positions in the reorder buffer change only when an instruction is committed. If the reorder buffer fills, we simply stop issuing instructions until an entry is made free.

Let's now examine how this scheme would work with the same example we used for Tomasulo's algorithm.

```
LD    F6, 34(R2)
LD    F2, 45(R3)
MULTD F0, F2, F4
SUBD  F8, F6, F2
DIVD  F10, F0, F6
ADDD  F6, F8, F2
```

Assume three adders and two multipliers, and the following latencies for the floating-point functional units: Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Let's look at how the reservation status tables look like when the `MULTD` is ready to go to commit.

Instruction status				
Instruction	Issue	Execute	Write result	Commit
`LD F6, 34(R2)`	✓	✓	✓	✓
`LD F2, 45(R3)`	✓	✓	✓	✓
`MULTD F0, F2, F4`	✓	✓	✓	
`SUBD F8, F6, F2`	✓	✓	✓	
`DIVD F10, F0, F6`	✓	✓		
`ADDD F6, F8, F2`	✓	✓	✓	

Reservation station status							
Name	Busy	Opcode	Vj	Vk	Qj	Qk	Dest
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	`MULTD`	Mem[45+Regs[R3]]	Regs[F4]			#3
Mult2	Yes	`DIVD`		Mem[34+Regs[R2]]	#3		#5

Reorder buffer.					
Entry	Busy	instruction	State	Destination	Value
1	No	`LD F6, 34(R2)`	Commit	F6	Mem[34+Regs[R2]]
2	No	`LD F2, 45(R3)`	Commit	F2	Mem[45+Regs[R3]]
3	Yes	`MULTD F0, F2, F4`	Write result	F0	#2 $\times$ Regs[F4]
4	Yes	`SUBD F8, F6, F2`	Write result	F8	#1 - #2
5	Yes	`DIVD F10, F0, F6`	Execute	F10	
6	Yes	`ADDD F6, F8, F2`	Write result	F6	#4 + #2

Register result status									
	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	3			6	4	5			
Busy	Yes	No	No	Yes	Yes	Yes	No	...	No

Note that although the `SUBD` instruction has completed execution, it does not commit until the `MULTD` commits, since commits happen in order. Note that all tags in the Qj and Qk fields as well as in the register status fields have been replaced with reorder buffer numbers, and the Dest field designates the reorder buffer number that is the destination for the result. This example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling: no instruction after the earliest uncompleted instruction, which is `MULTD`, is allowed to complete. In contrast, in the Tomasulo case, the `SUBD` and `ADDD` instructions would have also completed and written their results to the register file.

Reservation station status							
Name	Busy	Opcode	Vj	Vk	Qj	Qk	Dest
Mult1	No	`MULTD`	Mem[0+Regs[R1]]	Regs[F2]			#2
Mult2	No	`MULTD`	Mem[0+Regs[R1]]	Regs[F2]			#7

Reorder buffer.					
Entry	Busy	instruction	State	Destination	Value
1	No	`LD F0, 0(Rl)`	Commit	F0	Mem[0+Regs[R1]]
2	No	`MULTD F4, F0, F2`	Commit	F4	Regs[F0] $\times$ Regs[F2]
3	Yes	`SD 0(Rl), F4`	Write result	0+Regs[R1]	#2
4	Yes	`SUBI Rl,Rl,8`	Write result	R1	R1-8
5	Yes	`BNEZ Rl, Loop`	Write result		
6	No	`LD F0, 0(Rl)`	Write result	F0	Mem[#4]
7	No	`MULTD F4, F0, F2`	Write result	F4	#6 $\times$ Regs[F2]
8	Yes	`SD 0(Rl), F4`	Write result	0+Regs[R1]	#7
9	Yes	`SUBI Rl,Rl,8`	Write result	R1	#4-8
10	Yes	`BNEZ Rl, Loop`	Write result		

Register result status									
	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	6		7						
Busy	Yes	No	Yes	No	No	No	No	...	No

A processor with a reorder buffer can dynamically execute code while maintaining a precise interrupt model. For example. if the `MULTD` instruction caused an interrupt, we could simply wait until it reached the head of the reorder buffer and take the interrupt, flushing any other pending instructions. Because instruction commits happened in order, this yields a precise exception. By contrast, when using Tomasulo's algorithm, the `SUBD` and `ADDD` instructions could both complete before the `MULTD` raised the exception. The complication is that the registers F8 and F6 (destinations of the `SUBD` and `ADDD` instructions) could be overwritten, and the interrupt would be imprecise.

As another example, consider the following code snippet:

```
Loop: LD    F0,0(Rl)
      MULTD F4, F0, F2
      SD    0(Rl), F4
      SUBI  Rl, Rl, #8
      BNEZ  Rl, Loop     ; branches if Rl != 0
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the `LD` and `MULTD` from the first iteration have committed and all other instructions have completed execution. The above table shows the result.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that in the above example, the branch BNEZ is not taken the first time. The instructions prior to the branch will simply commit when each reaches the head of the reorder buffer: when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path. In speculative processors, performance is more sensitive to the branch prediction mechanisms, since the impact of a misprediction will be higher. Thus, all the aspects of handling branches—prediction accuracy, misprediction detection, and misprediction recovery—increase in importance.

Exceptions are handled by not recognizing the exception until the instruction is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the reorder buffer. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the reorder buffer is cleared. If the instruction reaches the head of the reorder buffer, then we know it is no longer speculative and the exception should really be taken.

Finally, although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units. Indeed, speculation may be more useful in integer programs since such programs tend to have code where the branch behavior is less predictable.